2023

Mapping and Navigation with the Fetch Robot

Dorothy Najjuma Kamya , '23

ENGR 90

Mapping and Navigation with the Fetch Robot

By: Dorothy Najjuma Kamya

Advisor: Professor Matthew Zucker

Table of Contents

Abstract

The goal of this project was to design software for the Fetch mobile manipulator robot to create a map of the second floor of Singer Hall, be able to locate itself on the map generated and navigate to a given position on the map. This was implemented within the Gazebo simulator initially, and then uploaded to the robot. The project can be extended to make the Fetch robot an interactive tour guide for Singer Hall, or to perform low-level tasks around the building like calling the elevator or carrying small objects between rooms. The interactive tour guide functionality is a particularly suitable application because the Fetch robot has audio capabilities and a robotic arm that make it more engaging than most robots like the turtlebot. This project was inspired after taking the Mobile Robotics course at Sarthmore.

Introduction

A lot of research has been carried out in the field of robotics towards autonomous robot navigation in unknown environments. One of the problems that has been worked on extensively is the Simultaneous Localization and Mapping (SLAM) problem [7]. For this problem, a robot is put in an unknown environment and made to navigate the environment while constructing or updating a map of its surroundings and simultaneously keeping track of its location within the map. A number of algorithms have been proposed to solve this problem, like the Extended Kalman Filter SLAM (EKF-SLAM), Graph SLAM and FastSLAM [7]. There are many variations of localization and mapping algorithms because the algorithms heavily depend on the robot sensor suite and map representations. Additionally, given a map of its environment, a robot can locate itself on the map using the adaptive Monte-Carlo localization technique, which uses an adaptive particle filter to make a probabilistically accurate guess of the robot's position on the map given input from the robot's sensors.

Robot Operating System (ROS) Overview

Interaction with the Fetch robot drivers and sensors is possible through the Robot Operating System (ROS). ROS is a set of software libraries and tools that make it easier to write robot applications. It provides a framework for interacting with the robot drivers and hardware. For this project, I used ROS because of its compatibility with the Fetch robot and its ubiquity in robotics research. ROS is maintained by the Open Robotics Foundation, and has an active open-source developer community.

ROS also has a number of useful features, software packages and third-party tools that were crucial for this project, and these include the following:

ROS nodes and topics

One of the foundational units of ROS are nodes and topics, which are used for passing messages between different parts of the system. A node is an executable program that runs a process, and every node performs a single task. A topic is a packet of information that stores data on a given property of the robot. Nodes can communicate with each other, and can publish or subscribe to different topics to change or read the value stored in the topic.

Managing the nodes, there is a Master in ROS which provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services, and enables individual ROS nodes to locate and communicate with one another. In order to launch multiple nodes at once, the roslaunch service was also useful in my project.

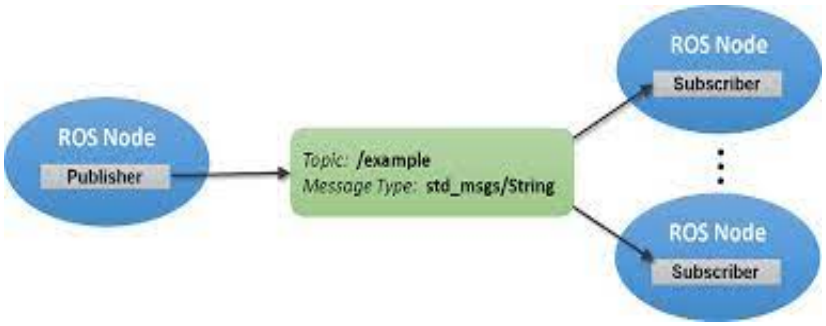

Figure 1: Diagram showing ROS Nodes and Topics. Sourced from Mathworks

Frames and Transforms

A robotic system typically has many 3D coordinate frames that change over time, such as a world/odometry frame, baselink frame and map frame. Transforms show the relationship between different coordinate frames. The tf package in ROS helps the user keep track of these multiple coordinate frames over time, maintaining the relationship between coordinate frames in a tree structure buffered in time. A user can transform points, vectors, etc between any two coordinate frames at any desired point in time, and use transforms to determine things like where the base of a robot is relative to the map frame. Transforms are an important part in navigation, especially when changing goal points defined in the map frame to the robot's base frame.

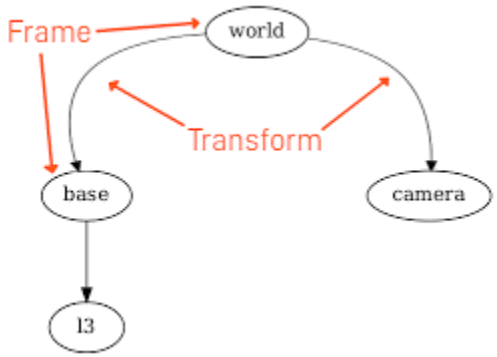

Figure 2: Diagram showing frames and transforms

Mapping and Localization Packages

ROS has a gmapping software package that provides laser-based Simultaneous Localization and Mapping using ROS nodes [3]. Using this package, one can create a 2-D occupancy grid map from laser and pose data collected by the mobile robot. The code implements the FastSLAM algorithm, which uses a highly optimized Rao-Blackwellized particle

filter to learn grid maps from laser range data. This deals better with errors and non-linearity compared to other mapping algorithms like EKF-SLAM.

Another ROS package that was useful for localization is the AMCL(adaptive monte-carlo localization) package [4]. AMCL is a probabilistic localization system for a robot moving in 2D. It implements adaptive Monte Carlo localization. This algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible location of the robot on the map. The particles become more condensed with increased certainty as the robot gets input from its sensors.

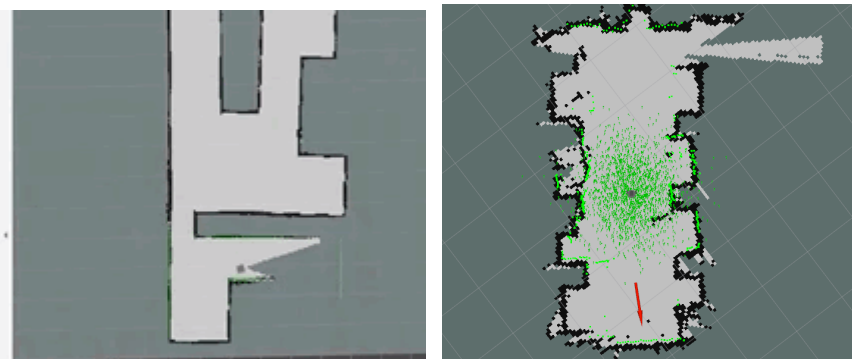

Figure 3: Diagrams showing map creation and AMCL

Navigation Stack and Actionlib

Other useful ROS packages include the navigation and actionlib packages. The navigation package contains a 2D Navigation Stack that takes in information from odometry and sensor streams and outputs velocity commands to send to a mobile base. The job of the navigation stack is to produce a safe path for the robot to execute, by processing data from sensors and the environment map. The move_base node within the Navigation Stack provides a

more advanced way to control a mobile robot - it is what does the work of determining the robot's position in the environment where it is located (localization) and planning and executing the path to a target location. It also enables autonomous avoiding of both static and dynamic obstacles based on continuous sensor inputs, specifically the input from the 2D laser scanner at the bottom, and the 3D depth scanner at the top.

The actionlib stack provides a standardized interface for sending preemptable tasks such as moving the base to a target location, performing a laser scan and returning the resulting point cloud, etc. The actionlib allows the user to send a request to a node to perform some task, and also receive a reply to the request. Commands can also be sent using ROS "services" but the actionlib provides tools to create servers that execute long-running goals with greater flexibility such as the ability to cancel the request and get periodic feedback on the request's status. To use the action lib, one creates an Action Client and Action Server which communicate via a "ROS Action Protocol" using ROS messages.

Simulation and Visualization Tools

Gazebo is a 3D simulation tool that is supported by ROS, complete with customizable rendered environments and a large selection of modeled robots including the turtlebot. Gazebo fully simulates robot use including models of sensors that "see" the simulated environment, such as laser scanners and cameras, and subscribes to ROS message topics so that a user can send the simulated root commands just like one would a real-world robot.

Rviz is a 3d visualization tool for ROS. It provides a wide selection of visualization options including a view of your robot model and captured sensor information from robot sensors such

as data from camera, lasers, point clouds, and maps such as those created by gmapping. Rviz can also be used for some minimal interfacing with your robot such as sending simple navigation goals using the 2D nav goal and providing initial estimates of the robot's location on a map using the 2D pose estimate.

Tkinter Package

Tkinter is the standard Python interface to the Tk Graphic User Interface (GUI) toolkit - an open-source, cross-platform widget toolkit that provides a selection of basic elements for building a GUI. Tkinter is independent of ROS, but can be used in coordination with the ROS framework. The tkinter package uses object oriented programming and wrappers that implement the interface widgets as Python classes. The perks of this package are that it's a relatively straightforward toolkit and well documented .

Project Design

This project involved designing navigational and user interface software systems that communicate with hardware. I used the ROS API, which is the list of ROS topics, services, action servers, and messages that a given robot is providing to give access to its hardware. ROS offers a standard software platform to developers across academia and industry, with core tools and libraries being maintained by OpenRobotics and ROS' Technical Steering Committee. Coding to this API conforms to an industry-wide standard of creating and interfacing with software in the field of robotics.

The applications I used like RViz and the Gazebo simulator follows industry-standard robotic platforms and create realistic environments with high fidelity sensor streams in order to produce consistent theoretical results in a way that is accurate to their real-world counterparts.

For this project, I adhered to common coding standards in the field of robotics. The code written will be relatively easy to understand for a fellow programmer and contain no redundancies or unnecessary repetitions, like multiple creations of the same ros node. This is to make it easier for engineering students to extend this project and add other functionalities to the Fetch robot in future projects.

Project Development

For this project, I started out with tutorials on navigation using a virtual Fetch robot in the Gazebo simulation. Simulation was an important step in the project since getting the mapping and navigation working in simulation gave me practice using the navigation packages, and made me more confident that I could generate similar results on the actual Fetch robot. The tutorials provided on Fetch's website, https://docs.fetchrobotics.com/ , on visualization tools, Gazebo, robot movement and navigation were instrumental in providing a starting point for my code base.

Once navigation was working in simulation, I attempted to connect to the Fetch robot using a virtual machine with Ubuntu 18 and Python2 installed. I used VirtualBox to launch the virtual machine and realized that certain network settings and ROS variables had to be set to be able to send and receive messages from the robot using a single ROS Master. The following had to be done for successful communication of ROS messages between the devices:

- Made an ssh connection linking the robot and the virtual machine.
- Made sure my computer and the robot were on the same network.
- Set the network settings of the virtual machine to "Bridged adapter" so that the IP address settings of the virtual machine matched those of my laptop.
- Set the ROS_IP variable to the virtual machine's IP address.
- Set the ROS_MASTER_URI variable to: http://robot_ip_address:11311

To implement the graphical user interface, online tutorials and documentation on the Tkinter library were helpful, especially the following:

- https://realpython.com/python-gui-tkinter/
- https://docs.python.org/2/library/tkinter.html

Results

Gazebo Simulation

Early on in the project, I implemented map creation and navigation in the Gazebo simulator. In Gazebo, I launched a virtual environment and robot and created a map of the virtual world. This used the navigation packages mentioned previously. To navigate to different places on the map, Rviz's 2D Nav Goal was used. All the code was run in the terminal.

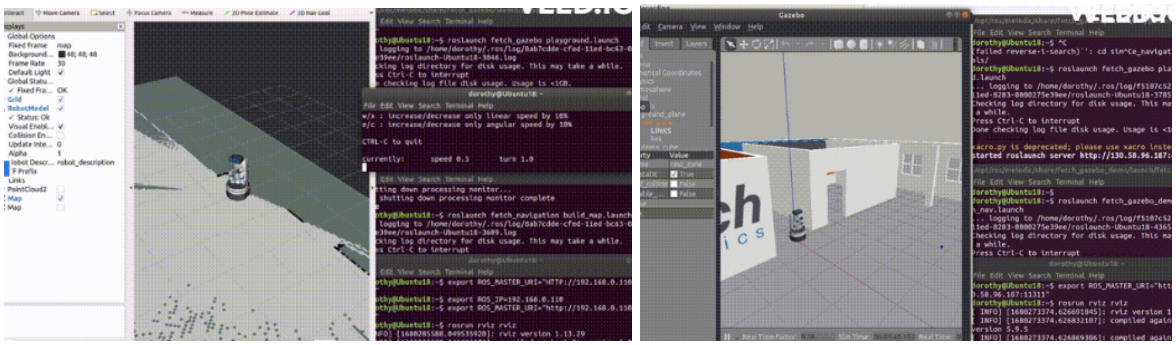

Figure 4: Diagram Showing Map Creation and Navigation in Gazebo Simulation

Work Done with the Fetch Robot

After successfully connecting my virtual machine to the Fetch robot as previously discussed, I navigated Fetch robot using a PS4 joystick to create a map of the second floor of

Singer Hall, visiting mainly the Engineering part of the floor. Rviz was used to visualize the map and make sure that the map created was as good as expected. The map was saved to be used for navigation. This map was very similar to the actual floor plan. After the map was created, I picked a few points for the robot to navigate by using Rviz's "Publish point" to get the point coordinates in the map frame. The points picked are shown in the generated map below:

Figure 6: Actual(top) and Generated(bottom) maps of Singer 2nd Floor

I then implemented a simple action client in Python to take user input on what goal to navigate to. The simple action client is a special kind of node that can send goals one-by-one to the navigation stack to be executed. The Python script I wrote also got feedback on the goal

status to publish back to the user to give more information on goal execution. The structure of my code and its interaction with major nodes is shown in the diagram below:

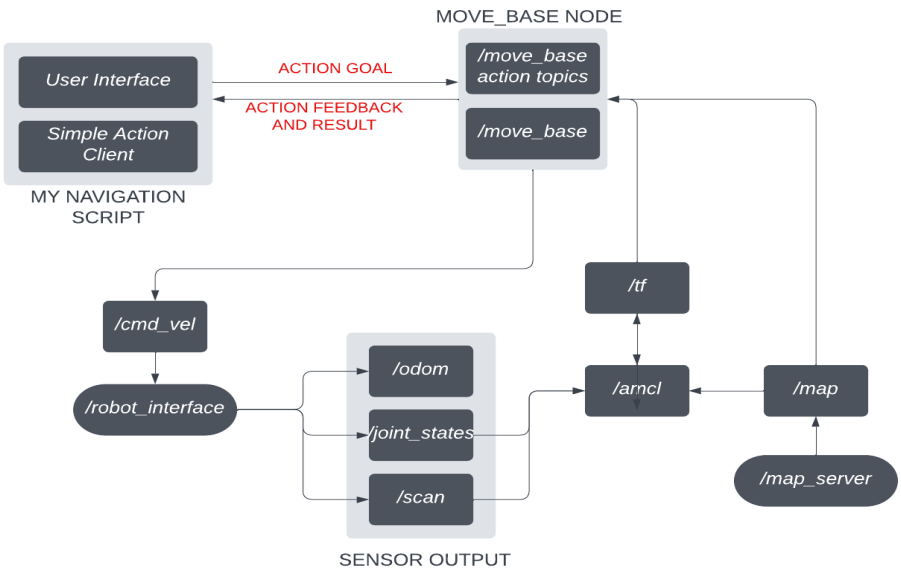

Figure 7: Diagram showing the structure of my program and major nodes

User Interface

After getting navigation working on the robot, I implemented a user interface using the in-built Tkinter library in Python. The interface uses the combobox widget to give the user a choice over a number of destinations to navigate to. After goal execution, the status is returned as a message in a pop-up window.

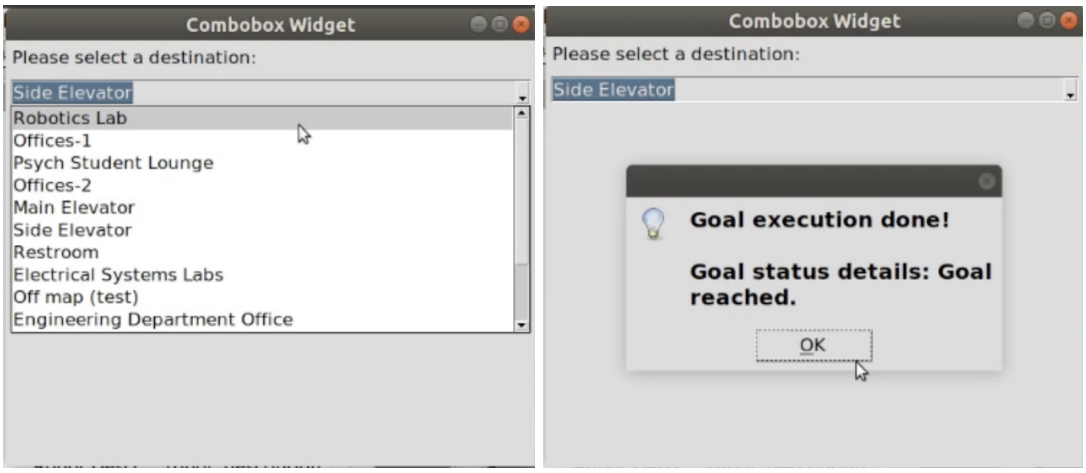

Figure 8: Diagram showing the User Interface Implemented when my program runs

Roslaunch

After implementing the user interface, I used roslaunch to launch all the nodes and services I needed to run for my project. I wrote a launch file that runs my python script and launches the user interface and simple action client written. The launch file also runs the navigation stack and launches Rviz with a pre-determined configuration for visualization purposes as the robot moves around on the map.

Figure 9: Image of the Fetch Robot Moving to Side Elevator in Singer

Conclusion

Over the course of the project, I learnt a lot from online tutorials as I developed my code. Trying to replicate tutorials and extending them formed the basis of my code. I also noticed most of the online sources use Python 3, which the Fetch robot has not yet been adapted for. I had to adapt my code to Python 2. I was able to build skills in ROS, Python and software project management through creating my own ROS packages. I also built my debugging skills and gained experience with networking, especially setting up an ssh connection between machines. Additionally, moving the robot around and seeing it navigate autonomously made the project very exciting!

Overall, the work completed towards this project met the criteria for success outlined in my proposal, which involved map creation and navigation in the Gazebo simulator and with the Fetch robot in the robotics lab, and a user-friendly interface to take in navigation goals.

Future work

- Adding robot arm functionality: The Fetch robot arm has many joints and multiple degrees of freedom, which makes arm motion a hard problem to solve because of the large parameter space that has to be optimized over. A useful package for arm manipulation is the MoveIt package in ROS. There are also existing tutorials and documentation to get started on this on the Fetch website .
- Mapping other floors of Singer hall so that the robot can navigate through the entire building: one would also need to have some type of identifier for each floor so that the robot can know which map to use based on which floor it is on.

- Extending the User Interface: A web-based interface with more graphics would be an improvement on my current interface.

Acknowledgements

A lot of appreciation goes to my project advisor, Professor Matt Zucker, with Swarthmore's Engineering Department, and my interim advisor, Professor Stephen Phillips.

References

[1] Amos, D. (2022, March 30). *Python GUI programming with Tkinter.* Real Python. Retrieved December 18, 2022, from https://realpython.com/python-gui-tkinter/

[2] Ferguson, M. (2016, March 6). *Wiki*. pocketsphinx. Retrieved December 18, 2022, from http://wiki.ros.org/pocketsphinx

[3] Gerkey, B. (2019, February 4). *Wiki*. ros.org. Retrieved December 18, 2022, from http://wiki.ros.org/gmapping

[4] Gerkey, B. P. (2020, August 27). *actionlib*. ros.org. Retrieved December 18, 2022, from http://wiki.ros.org/amcl

[5] J. Knoll, K. Hevrdejs, and S. Miah, “Virtual robot experiments for navigation in structured environments,” in The 26th International Symposium on Industrial Electronics, Edinburgh, Scotland, June 2017.

[6] Wikimedia Foundation. (2022, December 14). *A* search algorithm*. Wikipedia. Retrieved December 18, 2022, from https://en.wikipedia.org/wiki/A*_search_algorithm

[7] Wikimedia Foundation. (2022, November 8). *Simultaneous localization and mapping*. Wikipedia. Retrieved December 18, 2022, from https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping

Appendix: Code

Roslaunch file:

```
 <include file="$(find fetch_navigation)/launch/fetch_nav.launch">
        <arg name="map_file"
value="/home/dorothy/Desktop/Maps/SingerSecondFloor.yaml"/>
  </include>

  <node type="rviz" name="rviz" pkg="rviz" args="-d
/home/dorothy/Desktop/fetch_robot_config.rviz" />

  <node pkg="simple_navigation_goals" name="move_base_node"
type="move_base_node.py"  output="screen" launch-prefix="gnome-terminal
--command">
  </node>
```

Python Script:

```
#!/usr/bin/env python
# license removed for brevity

import rospy
# Brings in the SimpleActionClient
import actionlib
# Brings in the .action file and messages used by the move base action
from actionlib_msgs.msg import GoalStatus
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from sound_play.libsoundplay import SoundClient

import Tkinter as tk
import tkMessageBox
#from Tkinter import ttk #ttk,tkMessageBox is its own package separate from Tkinter in
python2
```

```
import ttk

def movebase_client(point,client):
  # Pass an action client called "move_base" with action definition file
"MoveBaseAction"

  # Creates a new goal with the MoveBaseGoal constructor
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()

        goal.target_pose.pose.position.x = point[0]
        goal.target_pose.pose.position.y = point[1]
  # No rotation of the mobile base frame w.r.t. map frame
        goal.target_pose.pose.orientation.w = 1.0

  # Sends the goal to the action server.
        client.send_goal(goal)
  # Waits for the server to finish performing the action.
        wait = client.wait_for_result()
  # If the result doesn't arrive, assume the Server is not available
        if not wait:
        rospy.logerr("Action server not available!")
        rospy.signal_shutdown("Action server not available!")
        else:
        # Result of executing the action
        return client.get_result(),client.get_goal_status_text()

def move_to_goal(dest,options,client,event,soundhandle):
   #sends goal to move_base action client and waits for goal navigation results
   try:
         goal = dest
         print("Goal Point: ",options[goal])
```

```
            result,goal_status = movebase_client(options[goal],client)
            if result:
                    rospy.loginfo("Goal execution done!")
                    print("Goal status details: %s "%(goal_status))
                    show_msg = ("Goal execution done! \n\nGoal status details: %s
"%(goal_status))
                    # add sound
                    soundhandle.stopAll()
                    if goal_status == "Goal reached.":
                            soundhandle.say("Success! We have arrived at the "+goal)
                    else:
                            soundhandle.say("Oh no! Failed to arrive at the "+goal)
                    tkMessageBox.showinfo(message=show_msg)
    except rospy.ROSInterruptException:
                    rospy.loginfo("Navigation test finished.")

def interface(options,client,soundhandle):
    # Creates a graphical user interface to use
    root = tk.Tk()
    # config the root window
    root.geometry('500x400') #widthxheight
    root.resizable(True, True)
    root.title('Combobox Widget')

    # label
    label = ttk.Label(text="Please select a destination:")
    label.pack(fill=tk.X, padx=5, pady=5)

    # create a combobox
    selected_dest = tk.StringVar()
    dest_cb = ttk.Combobox(root, textvariable=selected_dest)
    dest_cb['values'] = [key for key in options]

    # prevent typing a value
```

```
    dest_cb['state'] = 'readonly'

    # place the widget
    dest_cb.pack(fill=tk.X, padx=5, pady=5)

    # bind the selected value changes
    def dest_changed(event):
            tkMessageBox.showinfo(
             title='Result',
             message='You selected '+str(selected_dest.get())+'!'
            )
            if selected_dest.get():
            move_to_goal(selected_dest.get(),options,client,event,soundhandle) # send goal
to node

    dest_cb.bind('<<ComboboxSelected>>', dest_changed)
    root.mainloop()


# If the python node is executed as main process (sourced directly)
if __name__ == '__main__':
      try:
            options = {"Robotics Lab":[-3.12,-6.65,0], "Engineering Classes":[-9.35,-8.85,0],
"Side Elevator": [-0.93,-9.72,0], "Student Lounge":[15.4,-4.43,0], "Offices-1": [22,-1.62,0],
"Offices-2": [33.4,0.794,0], "Main Elevator": [42.3,-6.8,0], "Engineering Department
Office": [42.2,-0.317,0], "Restroom":[53.9,-9.8,0], "Psych Student Lounge":[51.3,-28,0],
"Electrical Systems Labs":[76.6,-1.29,0], "Off map (test)": [100,100,0], "In Lab":
[-1.85,0.09,0], "In Lab2": [-1.89,-3.29,0]}

        rospy.init_node('movebase_client_py') #initialize ros node
        client = actionlib.SimpleActionClient('move_base',MoveBaseAction)
        # Waits until the robot server has started up
        client.wait_for_server()
        # add sound
```

```
        soundhandle = SoundClient()

        interface(options,client,soundhandle) # run GUI to choose goals

except rospy.ROSInterruptException:
    rospy.loginfo("Navigation test finished.")
```